Understanding and analyzing software architecture (of distributed systems) using patterns

Stal, Michael

5 Service-Oriented Architecture Principles and Technologies

Has been published as: Using Architectural Patterns and Blueprints for Service-Oriented Architecture.

In: IEEE Software, Volume 23, No 2, March/April 2006

Author: Michael Stal

5.1 Abstract

Using software patterns and blueprints to express a service-oriented architecture's fundamental principles supports the efficient use of SOA technologies for application development.

5.2 Context

If you're a software expert, you've almost certainly encountered the topic of service-oriented architecture in recent years. However, most publications fail to explain SOA or simply assume it merely defines a synonym for a stack of XML Web service protocols and standards. In fact, there are many possible views of SOA—most of which focus on technologies and implementations for service orientation rather than the architecture. SOA in its fundamental core doesn't simply define an implementation technology but an architectural solution for a specific design problem in a given context—with XML Web services being just one possible implementation technology.

Understanding SOA and all of its implications for software applications requires introducing a set of architectural principles that define SOA more concretely. Once developers understand the SOA paradigm from an architectural perspective, they can better leverage SOA implementations.

As with many other technologies for developing distributed systems, ensuring transparency—hiding the underlying communication structure's implementation details—helps developers focus on domain-specific problems. However, many forces in addition to transparency drive service-oriented applications. To build software applications that not only consider the functional specification but also meet operational and developmental properties, developers must understand the underlying architectural principles.

So, how can we effectively define and express SOA's core principles? Software patterns are perfect for this. Contrary to common belief, software patterns are useful not only for building new software applications but also for understanding existing applications and their platforms. Patterns don't completely cover all general-purpose or domain-specific areas, but architectural blueprints—which aren't full-blown patterns but reveal the same properties — can fill the remaining gaps. Furthermore, software patterns and blueprints can accommodate both forward and reverse engineering.

Here, I illustrate both directions using SOA as an example. Using the core SOA principles I'll introduce, software architects can derive best practice pattern systems and catalogs that illustrate how to leverage existing SOA technologies.

5.3 Driving forces

The central objective of a service-oriented approach is to reduce dependencies between "software islands," which basically comprise services and the clients accessing those services.

These service-oriented software systems need to balance the following forces:

5.3.1 Distribution

From a logical perspective (but not necessarily in the physical implementation, since some layers and components might be collocated), the software environments under consideration consist of different software entities running on different network nodes that might need to cooperate via a communication protocol.

5.3.2 Heterogeneity

The distributed software entities typically reside in heterogeneous environments, so client developers can't control remote services' implementation details. Also, service developers can't assume a priori which kinds of clients will use the services and in which contexts.

5.3.3 Dynamics

The software systems mostly comprise highly dynamic environments, so designers can't statically predefine many decisions, because the decisions must be dynamically configured at runtime.

5.3.4 Transparency

Remote-services providers and consumers should be oblivious to the underlying communication infrastructure's implementation details. We can also derive transparency from the heterogeneity and dynamics.

5.3.5 Process-orientation

Services often implement fine-grained functionality, while clients need to compose services that result in more coarse-grained building blocks. So, it's essential to compose multiple services for coordinated workflows.

5.3.6 Implications

According to existing literature, these forces help drive loosely coupled systems [44]. I thus describe the SOA mantra as loose coupling. To balance these forces and enable loosely coupled software applications, the underlying distribution and communication infrastructure must follow architectural principles that support these forces.

5.4 Architectural Principles

5.4.1 Forces

Many publications define SOA using a ternary relationship model that depicts the main SOA participants and their dependencies. Service providers register their services with a central repository, and service consumers query the repository for the services they need.

Once clients have identified the right services within the repository, they can directly interact with those services.

This generic model doesn't explain the differences between standard middleware and the service-oriented approach. It turns out that the model applies to all kinds of distribution middleware including CORBA, a Distributed Component Object Model, Java remote method invocation (RMI), and .NET Remoting.

Refining this model for an SOA context requires defining the following aspects in more detail:

- *Interfaces and contracts*: How can service providers describe the services and contracts offered to the client? How can clients understand and access these services?

- *Communication*: Which communication styles are available for client-services interaction? What is the transmitted information's content and semantics?

- *Service lookup and registration*: How can service providers make their services known to clients, and how can clients locate the services they require?

- *State and activation*: How does the infrastructure deal with state information and activation issues, especially when it uses stateless communication protocols and services internally?

- *Processes and their implementation*: If clients must combine different independent services to complete processes, how can the infrastructure support service coordination and orchestration?

I can answer these questions using the architectural principles to which an SOA must conform. (I could cover other issues here, such as versioning and security concerns, but instead I spend more time elaborating on the architectural principles.)

5.4.2 Interfaces and contracts: Loose coupling

To decouple clients from service implementation details and decouple services from clients, the *Bridge pattern* [33] separates the service interface (see the *Explicit Interface pattern [13]*) from the service implementation (see Figure 18). Clients access an explicit

service interface, which delegates all incoming client requests to the actual service implementation.

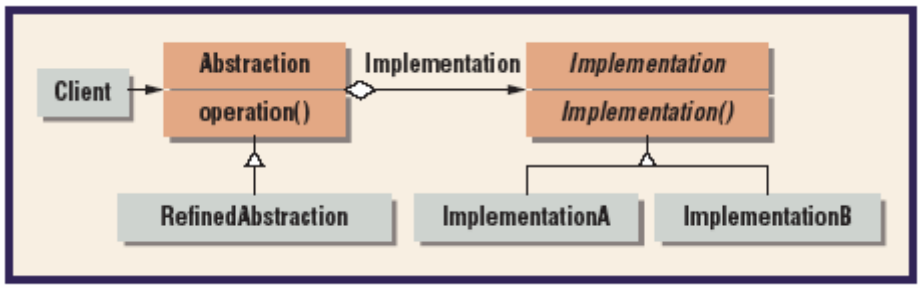

Figure 18: The Bridge pattern decouples clients and service implementations using interfaces as abstraction layers (italics indicate an abstract class).

So, you can change the service interface without changing the service implementation, and vice versa. Interfaces aren't necessarily limited to method invocation interfaces, at least not on the physical layer. They might also be signal-, message passing-, or event-based.

On the client side, the *Proxy pattern* [33][14] shields clients from all communication activities with the service, thus enabling distribution and transparency. If protocol independence is important, we might further refine the proxy implementation using the Forwarder-Receiver pattern [14].This pattern introduces forwarder components (which forward messages to a remote peer) and receiver components (which receive the messages from the remote peer) as additional participants for hiding the underlying communication infrastructure's details. It might also help foster dynamics by flexibly loading forwarders and receivers at runtime.

Describing and implementing interfaces are important with respect to heterogeneity. If clients and services can use different implementation technologies, developers must be able to define and express interfaces in a technology-agnostic way. Thus, clients and services must stick with a least common denominator. Proxies and bridges must provide a type mapping layer for marshaling and demarshaling data types between heterogene-

ous clients and services as well as between the communication layer and programming language at runtime.

An interface description language (which denotes a kind of Domain Specific Language) allows an explicit definition of the concrete contract between a service and its clients. The constituents of this contract might include

- the description of the service's functionality as well as preconditions, post conditions, and invariants;
- the service's physical location;
- a semantic description of the service; and
- quality-of-service and contextual information.

The *Reflection pattern* can help supply and leverage this kind of meta-information [14]. The *Interpreter pattern* [33] supports the development of generators that use the Reflection pattern to dynamically reflect over the interface descriptions and use the information to generate all necessary artifacts such as proxies or bridges either statically or dynamically.

5.4.3 Communication: Message-exchange patterns

To provide loose coupling and flexibility with respect to communication, an SOA must support various communication styles — for example, one-to-one and many-to-many communication and event- and remoting-based communication. On the bottom layer, clients and services communicate asynchronously by exchanging messages using various message exchange patterns. In the *Message Passing blueprint*, clients and services communicate by transmitting messages. Consequently, communicating peers must establish a direct communication channel to send messages to their remote peers.

However, for high flexibility and dynamics, communicating peers shouldn't transmit messages across a fixed communication line but rather through dynamic message routes. Applying the *Store-and-Forward blueprint* (or *Message Queue blueprint*; see http://www.michael-stal.com) introduces additional queues to store messages temporarily, forwarding them when the next receiver becomes available. So, no direct communication link exists between the client and service, and no tight coupling occurs between communication partners. (The detailed concepts of message-based communication appear elsewhere [41]).

When heterogeneous software entities send messages back and forth, it's essential that peers agree on a common format for the message packets (also called *documents*). Otherwise, the receiver can't interpret and understand the message. Two approaches exist for determining the format. In the first approach, for each communication relation, the peers individually agree on a concrete message structure and semantics. This approach leads to a combinational explosion if many peers are available. Additionally, applications must manually process messages.

In the second approach, a standardized specification defines the message content and semantics. This is less flexible but more convenient, because it allows automating message processing.

We can divide documents into a body that contains the actual message and an optional part that carries additional information such as quality-of-service properties or routing information.

Interceptors [85] are available on the client and on the server side responsible to automatically and transparently create and process these additional out-of-the-band properties such as security information, other quality-of-service properties, and service-level agreement information.

The communication protocol can support sending documents as binary units or plain text (for example, XML). Concrete SOA implementations might use both options.

Because transparency represents a driving SOA force, implementing a message exchange in the infrastructure's communication-specific parts doesn't imply that clients and services must interoperate in a message-oriented way. When system developers place the *Broker pattern* [14] above the message-based communication protocol, client- and server-side proxies and bridges hide communication internals and provide a synchronous invocation-based view of communication. In this case, additional patterns help deal with asynchronous messaging idiosyncrasies.

For example, the *Asynchronous Completion Token pattern* [85] applies to an associate reply with request messages by piggy-packing unique out-of-the-band tokens together with the messages themselves (so it includes additional information not related to the actual communication).

Patterns such as the *Observer pattern* [33] and the *Reactor pattern* [85] help deal with the asynchronous reception of reply messages.

5.4.4 Service lookup and registration

To access a specific service, a client must first locate it. Developers can hard-code location information in the client code, but this leads to tight coupling. It introduces location dependencies in client implementations, which results in additional liabilities such as reduced fault tolerance. A more flexible approach is to apply the *Client-Dispatcher-Server pattern* [14]— a constituent of the Broker pattern.

The Client-Dispatcher-Server pattern introduces an additional intermediate actor, the dispatcher, between clients and services. Service implementations register their services with the dispatcher, which acts as a service (information) repository. Clients query the dispatcher for available services. I can further improve this kind of loose coupling by add-

ing yet another indirection layer. Instead of maintaining service locations in its repository, the dispatcher might maintain interface descriptions to enable late binding strategies and flexible service discovery. For instance, if the same service is available from different implementations, a client might want to query the repository for additional information, such as quality-of-service aspects, to determine the most appropriate service implementation.

How clients specify services and their locations is also important. For example, clients might use opaque references, port numbers, or URLs to specify services, depending on the concrete communication protocol available.

Removing the clients' and services' dependency on the service repository's concrete location further improves loose coupling. So, the dispatcher turns into a proxy component that doesn't maintain service information locally but provides transparent access to remote repositories with the actual information. These repositories might be available at predefined locations, or clients could dynamically locate them.

For this purpose, the *Lookup pattern* [47] is a more reasonable choice, especially in the context of peer-to-peer networks.

5.4.5 State and activation: Services and singletons

Loose coupling and scalability increase when communication protocols and services don't maintain state information across multiple message exchanges. Service implementations may be provided as a single object (for example, using the *Resource Lifecycle Manager pattern* [47]). Different optimizations are possible when services and protocols are stateless, such as

- on-demand activation (*Activator pattern* - see chapter 7) or eviction of services (*Evictor pattern* [47]), which reduce resource contention, or

- preinstantiation of multiple service instances in an *object pool* [47] to improve service access time, especially when many clients are accessing the service.

Although the fundamental SOA infrastructure strives for statelessness, circumstances exist under which we must preserve the state and identity in client-service interactions:

- *Service affinity*: When services are associated with concrete entities, clients sometimes must address a concrete service instance that represents exactly a particular logical or physical entity.

- *Sessions*: For efficiency reasons, some services might provide information in several chunks or otherwise depend on the results of previous interactions. Thus, the service must maintain session information for its clients. Patterns such as *Caching* or *Lazy Acquisition* [47] are helpful here.

In both cases, the solution consists of providing additional context information that's transmitted in all message exchanges. We can again use the Asynchronous Completion Token pattern.

For service affinity, a token helps represent a specific service instance. The service implementation can use the token to dispatch a client request to the specified instance. Alternatively, the service singleton can use the token to assume a specific identity.

To maintain session information, the token can serve as a session database's primary key. Typically, the service uses the token as a primary key for a back-end database or file system that persistently retains the session information. But how can the service determine when the session ends? One possible option is to introduce leasing strategies [47].

So, the SOA implementation can manage state and identity information, or applications can manage the information manually. For transparency reasons, the first alternative

seems more appropriate. Of course, the infrastructure must beware of security issues such as token spoofing and reuse.

5.4.6 Processes and their implementation: Coordination and orchestration

In contrast to standard distribution middleware such as CORBA or Java RMI, an SOA implements processes as first-class entities.

Clients compose unrelated and independent services to processes to achieve a common goal that they couldn't reach using a single service.

For easy service composition, a language should be available that lets developers describe processes in a higher layer of abstraction instead of forcing clients to invoke all required services from within a standard programming language (using the *Language blueprint* [27]). From the process descriptions, a model-based generator creates new macro services that aggregate finer-grained services according to the process description.

Unfortunately, it's rarely sufficient to simply access a collection of existing services in an uncoordinated way. Often, the composition process must meet additional coordination requirements. Some scenarios require service compositions to behave in a transactional, atomic way—for example, if a critical service fails, the composition process must at least partially roll back the results of previous service invocations. The *Coordinator pattern* deals with this issue by introducing a central coordination instance [47]. Depending on concrete application requirements, the coordinator should apply the *Strategy pattern* [33] to let process initiators switch between different coordination strategies. For example, in the transaction example, the choice might be using a *Two-Phase Commit pattern* for transaction control or applying the *Compensating Transaction blueprint* [6]. The Asynchronous Completion Token pattern can enable this kind of service orchestration by providing a coordination context [85].

5.5 Concrete technology examples

The SOA paradigm is agnostic with respect to specific technologies and implementation options. Possible implementation technologies include XML Web services, Object Management Group CORBA, Java RMI, .NET Remoting, email, Message-Oriented Middleware (MOM), or TCP/IP. Even real-world mail services apply most of the architectural principles.

For the sake of brevity, I discuss only a MOM example here.

It's easy to apply MOM — such as the IBM MQSeries, Microsoft Message Queuing (MSMQ) services, or Java Message Service—when implementing the SOA paradigm.

5.5.1 Interfaces and contracts

In MOM, interfaces only comprise functionality for forwarding and receiving messages using simple message queues. Developers can implicitly specify service interfaces without involving the underlying middleware.

Messages have a predefined internal structure, but application developers can transfer deliberate content such as raw data, text, or streams within these messages. Method-based brokers can hide the underlying middleware's message-oriented nature. For example, in Microsoft COM+, queued components are implemented on top of MSMQ. As a consequence, each service interface is made explicit using the Component Object Model Interface Definition Language to specify the external interfaces of queued components. In other words, a particular domain-specific language is leveraged that maps domain-specific interface descriptions and service contracts to implementation artifacts that hide the underlying middleware from application developers.

5.5.2 Communication

MOM implementations are based on proprietary protocols. They provide a message passing approach whereby message producers send messages to either one or multiple receivers respectively to the queues with which they're associated. As already mentioned, the different participants can agree on the structure of the documents sent between message producers and consumers.

5.5.3 Service lookup and registration

Message senders explicitly address the physical location of a receive queue in which the MOM should place a message. Directory services usually store the queues' physical addressing information so that senders don't need explicit knowledge of the message receive queues.

5.5.4 State and activation

MOM implements services—that is, message receivers—as singletons. Session information can be provided in a message's infrastructure.

5.5.5 Processes and their implementation

Most MOM implementations support transaction-based programming but don't introduce the notion of processes explicitly. However, we could introduce a specific language such as the Business Process Execution Language to implement business processes in addition to MOM.

5.6 SOA Future Outlook

I've introduced the fundamental architectural principles that form the base of the current understanding of SOA systems. However, SOA technologies' instability and immaturity

make them fast-moving targets. Thus, the SOA paradigm itself is subject to further evolution.

Of course, no one can anticipate all of the elements of future service-oriented infrastructures, but here I identify some deficiencies of the current paradigm and illustrate possible solutions.

Currently, three main research questions exist:

- How can we leverage semantic information in SOA systems?
- How can we deal with integration issues when providing SOA frameworks?
- How can we extend the SOA approach with respect to the resources it supports?

5.6.1 Semantic integration

Most available SOA technologies emphasize a service-oriented system's syntactical aspects but don't sufficiently deal with semantic issues. Using semantic information, however, enables the development of adaptive and self configuring solutions. For example, semantic information could enable the automatic composition of service-based processes on behalf of clients and the dynamic integration of adapters if a particular client can't handle the service interface provided. So, SOA systems should integrate semantic information and semantic processing in different layers.

The semantic layer enriches the syntactic definition of services so semantic properties become an essential part of the service contract. Such information might include contextual information (for example, security roles or location information), quality-of-service attributes (such as response times), or ontology information (such as using keywords from standard taxonomies).

The communication layer then must understand and process the semantic information. Services that process semantic information, such as load balancers or security implementations, must be able to intercept client invocations, message exchanges, and service processing.

Finally, clients, services, and repositories need API support to introspect, express, or even change semantic information and behavior. The Reflection, Interpreter, and Interceptor patterns can help, as well as the *Decorator* and *Chain-of-Responsibility patterns* [33].

5.6.2 Middleware integration

Many toolkits for XML Web services separate aspects in different parts of their implementations, which is an adequate approach if it doesn't neglect transparency requirements.

Today, developers must manually integrate service implementations with back-end functionality, add horizontal functionality such as security, provide their own state and identity management, and deal with many other aspects of service invocation and processing in an implementation-specific manner. So, adding services to a given application increases complexity.

Moreover, many parts of an SOA application or infrastructure aren't easy to change, such as the underlying communication protocol or security mechanisms.

As a consequence, an SOA implementation shouldn't follow a monolithic approach but provide an integrated solution, where most parts are configurable and extensible without impacting client and service code. For example, it shouldn't matter to back-end developers if the internal J2EE components they're implementing are for Java RMI or for services that clients can invoke using SOAP.

The Reflection and Interceptor patterns as well as the *Container pattern* [106] can help introduce this kind of integrative SOA middleware (also known as the Enterprise Service Bus [15]).

5.6.3 Further resource types

Services in a service-oriented system denote functional building blocks that clients can remotely access in a technology-agnostic way. In addition to services, a distributed application typically must use other resource types, such as objects in a database or files in a file system. If the details of accessing these resource types should be hidden from applications (such as with Grid systems), a similar strategy applies. Developers can wrap resources such as files and persistent objects in a database as services.

5.7 Best-practice patterns

Applying patterns and blueprints to explain or invent infrastructures and paradigms is one side of the coin. The other side is leveraging this architectural knowledge to effectively develop efficient applications running on these infrastructures. Researchers have introduced best-practice pattern systems for Web-based applications for this reason, [3][58] and some pattern literature even focuses on SOA applications [25]. Here I apply some well-known Core J2EE patterns [3] that will help developers build efficient SOA applications.

First of all, it’s useful to defer some requirements from the SOA architectural principles with respect to application development.

For examples, systems should aim to

- minimize client-service communication, which is a time-consuming and resource intensive activity;

- decouple clients and services from infrastructural issues such as service discovery, actual communication, and implementation to achieve flexibility, loose coupling, and transparency; and

- increase application developer productivity.

Developers might apply the following patterns to achieve these goals (see Figure 19).

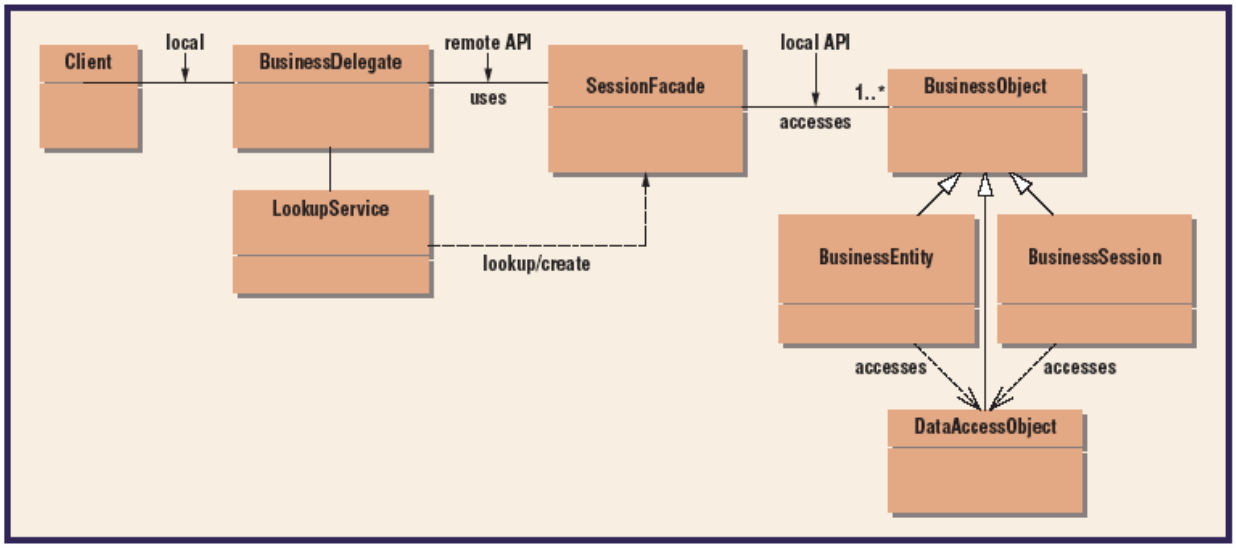

Figure 19: Applying core J2EE patterns in SOA based contexts.

A *Business Delegate* shields client applications from all aspects of remote communication with services such as discovery, message transfer, or exception handling. However, Business Delegates often need to perform the same tasks again and again, such as accessing the available service discovery mechanisms. A *Service Locator* and *Lookup Service* help move such common and iterative tasks to separate components.

Session Facades implement stateless and coarse-grained entities that trigger workflows and activities in the back end. Their interfaces should mainly offer functionality to initiate and manage whole processes. By not forcing clients to access multiple services directly, this pattern reduces communication overhead and decouples clients from any back-end changes such as workflow reorganizations.

Back-end functionality and services shouldn’t depend on low-level persistence mechanisms. Hence, *Data Access Objects* represent an intermediate layer that shields services and components from specifics of database and enterprise information systems.

To increase granularity of service interfaces, single message exchanges between clients and services shouldn’t transfer fine-grained data types but wrap complete semantically related data to *Transfer Objects*.

Although these patterns were originally used to build Java EE applications, we can apply them to SOA environments.

5.8 Summary

Using software patterns and blueprints to express an infrastructure or a technology’s fundamental principles applies a backward-engineering approach. In contrast to forward-engineering with patterns, this approach doesn’t require a complete pattern language or even a pattern system. However, knowing pattern variants and implementation options helps define how to change or extend specific architectural aspects, thus enforcing a paradigm’s systematic evolution. I applied this strategy to SOA by reverse-engineering fundamental architectural principles from existing SOA implementations. The approach is also an excellent vehicle for pattern mining and comparing alternative technology platforms.

Of course, this work only scratches the surface. We’ll need to dig deeper with future activities, such as introducing a systematic backward-engineering process as well as developing means to use the architectural principles discovered for model-based development and aspect-oriented programming.